# List of Links for this week:

<https://code.visualstudio.com/docs/editor/debugging>

<https://docs.microsoft.com/en-us/powershell/scripting/core-powershell/ise/how-to-debug-scripts-in-windows-powershell-ise?view=powershell-5.1>

<https://www.jetbrains.com/help/pycharm/debugging.html>

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.diagnostics/get-counter?view=powershell-6>

<https://technet.microsoft.com/en-us/library/ee872428.aspx>

# Debugging

Debugging is key to writing well-executed scripts. That’s why it’s important to become familiar with how the debugger for your IDE works.

For these examples, I’ll be using PowerShell ISE. This is a simple debugging program, but will allow us to try out some of the concepts of debugging.

Launch PowerShell ISE on your Windows machine.

Open the sysinfo.ps1 file we worked on a few weeks back. If this script isn’t available, you can copy it from <https://github.uc.edu/botheaj/it3038c-scripts/blob/master/powershell/sysinfo.ps1>

In ISE, right-click on the first line and select “Toggle BreakPoint”

Select “Run Script”

You should see right away that the breakpoint was hit. Now, using the Debug drop-down menu, we can slowly step through this program.

Notice how many steps the assignment of $IP is. Using ‘Step-in;, the debugger will step through every step that PowerShell takes to calculate this value.

We can use “Step-out” to skip the rest of this particular step and move on to the next step.

If we hover over $IP, we’ll see the value of that variable. This can be very useful when working with a script that has several variables.

Stepping over will skip a step and “Continue” will finish running the script.

Let’s try a loop in PowerShell and debug it. This loop will simply print 5 random numbers.

Open a new script and save it as ‘rando.ps1’

**$RANDO=0**

**for($i=0; $i -lt 5; $i++){**

**$RANDO=Get-Random -Maximum 1000 -Minimum 1**

**Write-Host($RANDO)**

**}**

Run it to make sure it works. Now, let’s debug. Set the breakpoint at the ‘for’ loop.

Notice how we can hover over all of the variable at any time and get the value for that variable at that point in time.

What may also be useful is writing output for every step of your script. This can be written to the console, or to a log file.

To do something like this, you can create a file and add content to that file as things happen.

**$RANDO=0**

**$Logfile = "C:\logs\rando.log"**

**for($i=0; $i -lt 5; $i++){**

**$RANDO=Get-Random -Maximum 1000 -Minimum 1**

**Write-Host($RANDO)**

**Add-Content $LogFile "INFO: Random number is ${RANDO}"**

**}**

Now open C:\logs\rando.log and view the results of the file. You may have to create the folder C:\logs.

Let’s stick to PowerShell a little longer. Debugging is something that comes with time, so to understand it, it’s best just to use it when the time arises and learn from that.

Let’s write a PowerShell script that will give us some performance data about our Windows machine. The Windows performance monitoring framework is known as Performance Logging and Alerting (PLA). PLA is built into Windows and uses COM and DCOM to obtain performance and diagnosis information from both local and remote computers. You can see PLA in action by opening the Task Manager | More Details | Performance | Open Resource Monitor.

Windows performance is measured in counters. There are counters for all aspects of the system, from memory to cpu to disk. We can see this data in PowerShell using the *Get-Counter* cmdlet.

**> Get-counter**

Get-counter returns the counter sets available as well as a sample of performance on this machine. Since we’re on a single machine, that’s what we’ll focus on, but you can easily send this command to multiple machines on your network to get performance data.

$Machines = ‘localhost’

Foreach ($machine in $Machines)

{

$RCounters = Get-Counter -ListSet \* -ComputerName $machine

“There are {0} counters on {1}” -f $RCounters.count, ($machine)

}

You can add multiple machines to the $Machines list to query them. The command tells us how many counters we have. We can use debugging to get a more in-depth look at how this ran. Get-Counter allows us to specify key pieces of data that we’re concerned about, like Processor, Memory, and Networking. We can specify these properties when we run our Get-Counter command. Let’s modify our script to specify some more key data. To do this, we’ll need to specify Get-Counter for each set of values we want.

$Machines = 'localhost'

Foreach ($machine in $Machines)

{

$pt = (Get-Counter -Counter "\Processor(\_Total)\% Processor Time").CounterSamples.CookedValue

$pt

}

This is helpful, but we need to get a sampling of data that we can use. To do this, we can add -SampleInterval 1 and -Max-Samples 10 to get 10 samples at one per second.

$Machines = 'localhost'

Foreach ($machine in $Machines)

{

$pt = (Get-Counter -Counter "\Processor(\_Total)\% Processor Time" -SampleInterval 1 -MaxSamples 10).CounterSamples.CookedValue

$pt

}

Now, let’s see if we can pretty this up a bit. We can see by debugging that we’re getting an array of objects back for $pt. We can do another Foreach loop and print them in a more readable fashion.

$Machines = 'localhost'

Foreach ($machine in $Machines)

{

$pt = (Get-Counter -Counter "\Processor(\_Total)\% Processor Time" -SampleInterval 1 -MaxSamples 3).CounterSamples.CookedValue

$sample = 1

foreach ($p in $pt) {

"Sample {2}: CPU is at {0}% on {1}" -f [int]$p, $machine, $sample

$sample++

}

}

If we want, we can write this to a file for later. Simply pipe this output to a file on your hard drive.

"Sample {2}: CPU is at {0}% on {1}" -f [int]$p, $machine, $sample | out-file C:\output.txt

Notice that it only wrote the last output. We need to include the -append flag to keep writing to it.

out-file -append C:\output.txt

It may also help to throw a date-time stamp in front. To do this, we’ll just edit our Sample output and include a formatted Get-Date.

$Machines = 'localhost'

Foreach ($machine in $Machines)

{

$pt = (Get-Counter -Counter "\Processor(\_Total)\% Processor Time" -SampleInterval 1 -MaxSamples 3).CounterSamples.CookedValue

$sample = 1

foreach ($p in $pt) {

$date = Get-Date -Format g

"{3} - Sample {2}: CPU is at {0}% on {1}" -f [int]$p, $machine, $sample, $date | Out-File -Append C:\output.txt

$sample++

}

}

Congratulations, you just created your first log file!

# Modules and Plugins

# List of Links for this week:

<https://www.powershellgallery.com/>

<https://pypi.org/>

<https://www.npmjs.com/>

<https://code.visualstudio.com/docs/editor/debugging>

<https://docs.microsoft.com/en-us/powershell/scripting/core-powershell/ise/how-to-debug-scripts-in-windows-powershell-ise?view=powershell-5.1>

<https://www.jetbrains.com/help/pycharm/debugging.html>

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.diagnostics/get-counter?view=powershell-6>

<https://technet.microsoft.com/en-us/library/ee872428.aspx>

Each programming language allows for a set of tools to ‘extend’ the functionality of the language. These go by several names, like plugins and modules. In this lab, we’ll explore how to use a module and what kind of benefits it can provide. We’ll also do a bit of debugging and logging to help troubleshoot our application.

# PowerShell Modules

A few weeks back, we wrote a function in PowerShell:

Function GetIP {

   $IP = Get-NetIPAddress | Where-Object {$\_.IPv4Address -like '192\*'}

   return $IP.IPv4Address

}

Let’s rewrite this function to a new file. Save this function in a file called ***tools.psm1***.

Save this file to your PowerShell folder.

Now before we do anything with this, let’s try running GetIP from your PowerShell command window (not from ISE), this command:

GetIP

You should get an error saying getip doesn’t exist. Now, let’s import our module file like so (note that your path may differ):

import-module C:\it3038c-scripts\powershell\tools.psm1

Now from the PowerShell command line, run GetIP again. This should return your host IP address.

GetIP

If you get an error, run the following command to correct the executionpolicy on your server

Set-ExecutionPolicy Unrestricted

Run getip one more time.

When it comes to code reuse, writing modules is vital. PowerShell V5 comes pre-built with module importing in mind. In fact, they’ve launched a whole website dedicated to module installation.

<https://www.powershellgallery.com>

Powershell gallery gives you access to all of the latest PowerShell modules available online. In fact, we can run a PowerShell command to find PowerShell modules

find-module

For this example, let’s install the Posh-SSH module, which will give us SSH abilities natively in PowerShell.

Find-Module Posh-SSH | Install-Module

Now, using our PowerShell knowledge, we can see all of the posh-ssh command now available to us.

Get-command -module Posh-SSH

Now, we can use this to SSH to our own Linux machine

New-SSHSession –ComputerName “192.168.33.XX” –credential (Get-Credential)

This simply establishes an SSH session. Make note of the index of the session that was created. We can now invoke remote commands from PowerShell.

Invoke-SSHcommand -Index 0 –Command “uptime”

This will return the uptime of our Linux machine. This is obviously not Native SSH on PowerShell, but for running quick remote commands from Windows to Linux, it’s a start.

***NOTE: Do not run this command from PowerShell ISE! PowerShell ISE has its own profile setting, so keep that in mind.***

One issue with PowerShell is that it doesn’t save our modules when we import them. We can fix this by editing our $profile. We first need to create this file, as it doesn’t exist by default:

new-item -path $profile -itemtype file -force

We can edit this file in the ISE.

powershell\_ise $profile

To load this script at every PowerShell startup, we simply need to call it.

Import-module C:\it3038c-scripts\powershell\tools.psm1

Save the script and restart PowerShell. Try to run getip again and it will work.

Use the $profile file to add any commands that you want to run at startup of PowerShell.

# Python Modules with Pip

Python can be extended using modules, typically installed using PIP.

Now, you can install a python module anywhere at anytime, however, if you’re not careful, you can make things very difficult for yourself if you decide to switch from one project to another.

To prevent this, Python created a plugin isolation method known as VirtualENV. VirtualENV allows you to set and install your Python modules to a specific directory, known as your virtual environment. You activate this VENV when you work with a specific set of scripts, and deactivate it when you’re finished. The benefit is that you can have certain versions of plugins, certain specific plugins, and plugin modifications that don’t impact the entire system.

***THESE COMMANDS ARE IN LINUX. IN WINDOWS, CALL ‘venv/webscraping/Scripts/activate.ps1’ to activate***

Let’s create our first VirtualENV. First, make sure virtualenv is installed.

$ pip install virtualenv

$ virtualenv ~/venv/webscraping/

Now, “ACTIVATE” this venv so we can demonstrate how this works.

Linux:

$ source ~/venv/webscraping/bin/activate

Windows:

$ ~/venv/webscraping/Scripts/activate.ps1

You can tell it’s activated if you see (**webscraping**) in front of your command line.

$ pip install bs4

Now, run the bs4 import command to verify it works

$ python

>>>import bs4

If you don’t get an error, then you know it works.

Now, deactivate your venv and rerun the bs4 import command

$ deactivate

$ python

>>>import bs4

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ModuleNotFoundError: No module named 'bs4'

The import command will no longer work, because we are no longer in our venv. You can re-activate it by re-running the ‘source’ command above.

SIDENOTE: pipenv will combine pip and virtualenv together, but for now, we’ll keep them separate for example sake.

# NodeJS Modules with NPM

Last up on our tour of modules is NodeJS. Node uses Node Package Manager (NPM) to manage its plugins. NPM is installed with Node. Unlike Python, all NPM modules are installed within the directory you’re working in, unless otherwise specified with the -g flag.

We installed a module last week to help us get the IP faster. Let’s install a helpful module called *node-dev*. Node-dev will watch the server.js file for changes and automatically restart the service when a change is detected. To install an npm module, use the following. From your ‘node’ folder:

$ npm install node-dev

Notice your folder now contains a ‘node\_modules’ folder. This contains the module and all of its dependencies.

For this particular module, I actually do want to install it globally, so that it can be used in any node folder. To do this, I’ll need sudo rights

$ sudo npm install -g node-dev

Now outside of our node directory, we can call the node-dev executable. Copy your server.js file from this location to /tmp, then cd to /tmp and run it with node-dev

$ cp server.js /tmp/server.js

$ cd /tmp

$ node-dev server.js

It works, because we installed this module globally.

# README.md

Real quick, the README.md is used for documentation. With Project 1, I didn’t really mark one way or another if you put all your code in a word doc, a comment section or in the README. For Projects 2 and 3, I’d like that documentation to be in the README.md. README.md is written in MARKDOWN language, which we touched a bit upon earlier this semester. Refer to this MARKDOWN cheat sheet <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>

# LAB 7

Using the language of your choice, research a plugin that you might find useful. To find a plugin, use google or any of these sites for each language:

PowerShell: <https://www.powershellgallery.com/>

Python: <https://pypi.python.org/pypi>

Node: <https://www.npmjs.com/>

Write a script that uses that plugin and include at least 3 different usages of the plugin that you’ve selected. For example, if you’ve selected to use Pillow, a Python plugin for manipulating images, give 3 different examples of tasks you can use this plugin for, like blurring an image, creating a thumbnail or applying a filter.

Once you’ve written a script that shows these examples, create a README.md that tells me how to install and run the script. You can view the example I’ve created at https://github.uc.edu/botheaj/it3038c-scripts/blob/master/README.md

**TIP: It’s actually easier to edit the README.md within the browser itself. Just create the file and push it to your github repo, then select it to edit it with GitHub. There is a “Preview changes” button that will give you a live preview of the changes you’re making. Make sure you *PULL* this change back down to your local repo after you’ve made them.**

# Submit to blackboard a link to your github with your example scripts and updated README.md